# Swinburne University of Technology

Faculty of Science, Engineering and Technology

## ASSIGNMENT COVER SHEET

---

**Subject Code:** COS30008
**Subject Title:** Data Structures and Patterns
**Assignment number and title:** 3, List ADT
**Due date:** Nov 3rd 2024, 23:59(Vn time)
**Lecturer:** Dr. Ky Trung

---

**Your name: Nguyen Thuan Khang_____** **Your student id:** 104171078

| Check Tutorial | Mon 10:30 | Mon 14:30 | Tues 08:30 | Tues 10:30 | Tues 12:30 | Tues 14:30 | Tues 16:30 | Wed 08:30 | Wed 10:30 | Wed 12:30 | Wed 14:30 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | |

---

Marker's comments:

| Problem | Marks | Obtained |
|---|---|---|
| 1 | 48 | |
| 2 | 28 | |
| 3 | 26 | |
| 4 | 30 | |
| 5 | 42 | |
| Total | 174 | |

---

**Extension certification:**

This assignment has been given an extension and is now due on _____

Signature of Convener:_____

## ListPS3.h:

```
// COS30008, List, Problem Set 3, 2024
#pragma once
#include "DoublyLinkedList.h"
#include "DoublyLinkedListIterator.h"
#include <stdexcept>
template<typename T>
class List
{
private:
	// auxiliary definition to simplify node usage
	using Node = DoublyLinkedList<T>;
	Node* fRoot; // the first element in the list
	size_t fCount; // number of elements in the list
public:
	// auxiliary definition to simplify iterator usage
	using Iterator = DoublyLinkedListIterator<T>;
	~List() // destructor - frees all nodes
	{
	while (fRoot != nullptr)
	{
	if (fRoot != &fRoot->getPrevious()) // more than one element
	{
	Node * lTemp = const_cast<Node*>(&fRoot->getPrevious()); // select last
	lTemp->isolate(); // remove from list
	delete lTemp; // free
	}
	else
	{
	delete fRoot; // freelast
	break; // stoploop
	}
	}
	}
		void remove(const T& aElement) // remove first match from list
	{
	Node * lNode = fRoot; // start at first
	while (lNode != nullptr) // Are there still nodes available ?
	{
	if (**lNode == aElement) // Have wefound the node ?
	{
	break; // stop the search
	}
	if (lNode != &fRoot->getPrevious()) // not reached last
	{
	lNode = const_cast<Node*>(&lNode->getNext()); // go to next
	}
	else
	{
	lNode = nullptr; // stop search
	}
	}
	// At this point we have either reached the end or found the node.
	if (lNode != nullptr) // We have found the node.
	{
	if (fCount != 1) // not the last element
	{
	if (lNode == fRoot)
	{
	fRoot = const_cast<Node*>(&fRoot->getNext()); // make next root
	}
	}
	else
	{
	fRoot = nullptr; // list becomes empty
	}
	lNode->isolate(); // isolate node
	delete lNode; // release node's memory
	fCount--; // decrement count
	}
```

```
	}
		////////////////////////////////////////////////////////////////
		//// PS3
		////////////////////////////////////////////////////////////////
		// P1

	List() : fRoot(nullptr), fCount(0) {} // default constructor

	bool empty() const
	{
		return fRoot == nullptr;
	} // Is list empty?

	size_t size() const
	{
		return fCount;
	} // list size

	void push_front(const T& aElement)
	{
		if (empty())
		{
			fRoot = new Node(aElement);
		}
		else
		{
			Node* lNode = new Node(aElement);
			fRoot->push_front(*lNode);
			fRoot = lNode;
		}
		++fCount;
	} // adds aElement at front

	Iterator begin() const
	{
		return Iterator(fRoot).begin();
	} // return a forward iterator

	Iterator end() const
	{
		return Iterator(fRoot).end();
	} // return a forward end iterator

	Iterator rbegin() const
	{
		return Iterator(fRoot).rbegin();
	} // return a backwards iterator

	Iterator rend() const
	{
		return Iterator(fRoot).rend();
	} // return a backwards end iterator
	// P2
	void push_back(const T& aElement) {
		if (empty())
		{
			fRoot = new Node(aElement);
		}
		else
		{
			Node* lastNode = const_cast<Node*>(&fRoot->getPrevious());
			lastNode->push_back(*new Node(aElement));
		}
		++fCount;
	} // adds aElement at back
	// P3
	const T& operator[](size_t aIndex) const {
		if (aIndex >= fCount) {
			throw std::out_of_range("Index out of range"); // Throw an exception if
index is out of range
		}
```

```
	Node* current = fRoot; // Start from the first element
	for (size_t i = 0; i < aIndex; ++i) {
		current = const_cast<Node*>(&current->getNext()); // Move to the next node
	}

	return **current; // Return the value of the element at the given index
}
// list indexer
// P4
List(const List& aOtherList) : fRoot(nullptr), fCount(0)
{
	*this = aOtherList;
} // copy constructor

List& operator=(const List& aOtherList)
{
	if (&aOtherList != this)
	{
		this->~List();
		if (aOtherList.fRoot == nullptr)
		{
			fRoot = nullptr;
		}
		else
		{
			fRoot = nullptr;
			fCount = 0;
			for (auto& payload : aOtherList)
			{
				push_back(payload);
			}
		}
	}
	return *this;
}
// P5
List(List&& aOtherList) : fRoot(nullptr), fCount(0)
{
	*this = std::move(aOtherList);
} // move constructor

List& operator=(List&& aOtherList)
{
	if (&aOtherList != this)
	{
		this->~List();
		if (aOtherList.fRoot == nullptr)
		{
			fRoot = nullptr;
		}
		else
		{
			fRoot = aOtherList.fRoot;
			fCount = aOtherList.fCount;
			aOtherList.fRoot = nullptr;
			aOtherList.fCount = 0;
		}
	}
	return *this;
} // move assignment operator

void push_front(T&& aElement)
{
	if (empty())
	{
		fRoot = new Node(std::move(aElement));
	}
	else
	{
		Node* lNode = new Node(std::move(aElement));
		fRoot->push_front(*lNode);
		fRoot = lNode;
```

```
            }
            ++fCount;
        } // adds aElement at front

        void push_back(T&& aElement)
        {
            if (empty())
            {
                fRoot = new Node(aElement);
            }
            else
            {
                Node* lastNode = const_cast<Node*>(&fRoot->getPrevious());
                lastNode->push_back(*new Node(aElement));
            }
            ++fCount;
        } // adds aElement at back
};
```